CS631 - Advanced Programming in the UNIX Environment

~

UNIX development tools

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/
HW1 revisited

- know your pre-defined constants (e.g., BUFSIZ, PATH_MAX, ...); see limits.h / limit(3)
- avoid code duplication in multiple blocks
- don't strcat(3) to argv[]
- the checklist is not there just to annoy you
Software Development Tools
Software Development Tools

```c
int rval;
int i;

/* Create socket */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    perror("opening stream socket");
    exit(1);
}

/* Name socket using wildcards */
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = 0;
if (bind(sock, (struct sockaddr *)&server, sizeof(server))) {
    perror("binding stream socket");
    exit(1);
}

/* Find out assigned port number and print it out */
length = sizeof(server);
if (getsockname(sock, (struct sockaddr *)&server, &length)) {
    perror("getting socket name");
    exit(1);
}
printf("Socket has port %d\n", ntohs(server.sin_port));

/* Start accepting connections */
listen(sock, 5);
do {
    msgsock = accept(sock, 0, 0);
    if (msgsock == -1)
        perror("accept");
    else do {
        bzero(buf, sizeof(buf));
        if ((rval = read(msgsock, buf, 1824)) < 0)
            perror("reading stream message");
        i = 0;
        if (rval == 0)
            printf("Ending connection\n");
        else
            printf("->-%s\n", buf);
    } while (rval != 0);
    close(msgsock);
} while (TRUE);
```
Software Development Tools

UNIX Userland is an IDE – essential tools that follow the paradigm of “Do one thing, and do it right” can be combined.

The most important tools are:

- `$EDITOR`
- the compiler toolchain
- `gdb(1)` – debugging your code
- `make(1)` – project build management, maintain program dependencies
- `diff(1)` and `patch(1)` – report and apply differences between files
- `cvs(1), svn(1), git(1)` etc. – distributed project management, version control
EDITOR

Know your $EDITOR. Core functionality:

- syntax highlighting
- efficient keyboard maneuvering
- setting markers, using buffers
- copy, yank, fold e.g. blocks
- search and replace
- window splitting
- autocompletion
- jump to definition / manual page
- applying external commands and filters
EDITOR

Examples given using `vim(1)`.

Efficient keyboard maneuvering:

- up, down, left, right (h, j, k, l)
- move by word, go to end (w, b, e)
- search forward, backward, move to beginning or end of line (, /, ?, ^, $)
- page up or down (^D, ^B)
- center page, top or bottom (zz, zt, zb)
- move to matching brace, move to beginning/end of code block (%, ]}, [{}]
- move through multiple files (:n, :prev, :rew)
EDITOR

Examples given using `vim(1)`.

Copy, yank, fold, markers, buffers etc.:

- set and display markers (m [a-zA-Z], :marks)
- select visual blocks (v, V)
- format / indent selected block (=)
- delete, yank, use of buffers (d, y, ”xy, ”xp)
- fold sections (zf, zA)
EDITOR

Examples given using `vim(1)`.

Look-ups:

- `find /usr/src -name '*[ch]' -print | xargs ctags -f ~/.ctags`
- `echo "set tags+=~/.ctags" >> ~/.vimrc`
- Ctrl+], Ctrl+t – jump to definition and back
- K – jump to manual page
- Ctrl+N – autocomplete
Examples given using `vim(1)`.

Integration with compiler, debugger, `make(1)` etc.

```
vim welcome.c
:make
Ctrl+]  
:cnext
...
```

Finally, two of your most powerful Unix IDE integrations are a terminal multiplexer (e.g. `screen(1)` or `tmux(1)`) and copious use of `Ctrl+Z` (i.e., the shell’s job control mechanisms).
Examples given using `vim(1)`.
Compilers

A compiler translates *source code* from a high-level programming language into *machine code* for a given architecture by performing a number of steps:

- lexical analysis
- preprocessing
- parsing
- semantic analysis
- code optimization
- code generation
- assembly
- linking
Compilers

- Language 1 source code
  - Compiler front-end for language 1
    - Lexical Analyzer (Scanner)
    - Syntax/Semantic Analyzer (Parser)
    - Intermediate-code generator
    - Non-optimized intermediate code
  - Intermediate code optimizer
    - Optimized intermediate code
  - Target-1 code generator
    - Target-1 machine code

- Language 2 source code
  - Compiler front-end for language 2
    - Lexical Analyzer (Scanner)
    - Syntax/Semantic Analyzer (Parser)
    - Intermediate-code generator
    - Non-optimized intermediate code
  - Intermediate code optimizer
    - Optimized intermediate code
  - Target-2 code generator
    - Target-2 machine code
Compilers

There are many different closed- and open-source compiler chains:

- Intel C/C++ Compiler (or icc)
- Turbo C / Turbo C++ / C++Builder (Borland)
- Microsoft Visual C++
- ...
- Clang (a frontend to LLVM)
- GNU Compiler Collection (or gcc)
- Portable C Compiler (or pcc)
- ...

Lecture 10: Things That Will Make Your Life Significantly Easier
The compiler toolchain

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).
Preprocessing

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```bash
$ cd compilechain
$ cat hello.c
$ man cpp
$ cpp hello.c hello.i
$ file hello.i
$ man cc
$ cc -v -E hello.c > hello.i
$ more hello.i
$ cc -v -DFOOD="Avocado" -E hello.c > hello.i.2
$ diff -bu hello.i hello.i.2
```
Compilation

The compiler chain or driver usually performs preprocessing (e.g. via \texttt{cpp(1)}), compilation (\texttt{cc(1)}), assembly (\texttt{as(1)}) and linking (\texttt{ld(1)})

\$ more hello.i
\$ cc -v -S hello.i
\$ file hello.s
\$ more hello.s
Assembly

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

$ as -o hello.o hello.s$
$ file hello.o$
$ cc -v -c hello.s$
$ objdump -d hello.o$
[...]
Linking

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

$ ld hello.o

[...]

$ ld hello.o -lc

[...]

$ cc -v hello.o

[...]

$ ld -dynamic-linker /usr/libexec/ld.elf.so
        /usr/lib/crt0.o /usr/lib/crti.o /usr/lib/crtbegin.o
        hello.o -lc /usr/lib/crtend.o /usr/lib/crtn.o

$ file a.out

$ ./a.out
Linking

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ cc -v -DFOOD="Avocado" hello.c 2>&1 | more
```
cc(1) and ld(1)

The compiler chain or driver usually performs preprocessing (e.g. via cpp(1)), compilation (cc(1)), assembly (as(1)) and linking (ld(1)).

Different flags can be passed to cc(1) to be passed through to each tool as well as to affect all tools.

$ cc -v -O2 -g hello.c 2>&1 | more
**cc(1) and ld(1)**

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

The order of the command line flags *may* play a role! Directories searched for libraries via `-L` and the resolving of undefined symbols via `-l` are examples of position sensitive flags.

```
$ cc -v main.c -L./lib2 -L./lib -lldtest 2>&1 | more
```

```
$ cc -v main.c -L./lib -L./lib2 -lldtest 2>&1 | more
```
cc(1) and ld(1)

The compiler chain or driver usually performs preprocessing (e.g. via cpp(1)), compilation (cc(1)), assembly (as(1)) and linking (ld(1)).

Different flags can be passed to cc(1) to be passed through to each tool as well as to affect all tools.

The order of the command line flags may play a role! Directories searched for libraries via -L and the resolving of undefined symbols via -l are examples of position sensitive flags.

The behavior of the compiler toolchain may be influenced by environment variables (eg TMPDIR, SGI_ABI) and/or the compilers default configuration file (MIPSPro’s /etc/compiler.defaults or gcc’s specs).

$ cc -v hello.c
$ TMPDIR=/var/tmp cc -v hello.c
$ cc -dumpspec
A Debugger
The purpose of a debugger such as `gdb(1)` is to allow you to see what is going on “inside” another program while it executes – or what another program was doing at the moment it crashed. `gdb` allows you to:

- make your program stop on specified conditions (for example by setting `breakpoints`)
- examine what has happened, when your program has stopped (by looking at the `backtrace`, inspecting the value of certain variables)
- inspect control flow (for example by `stepping` through the program)

Other interesting things you can do:

- examine stack frames: `info frame`, `info locals`, `info args`
- examine memory: `x`
- examine assembly: `disassemble func`
```
$ cc simple-ls.c
$ ./a.out ~/testdir
Memory fault (core dumped)
$ gdb ./a.out
(gdb) run ~/testdir

Program received signal SIGSEGV, Segmentation fault.
0x00000000000400cc7 in main (argc=2, argv=0x7f7fff6719f8) at simple-ls-stat.c:48
warning: Source file is more recent than executable.
48     printf("%s (%s)\n", dirp->d_name, pwd->pw_name);

(gdb) bt
(gdb) frame 0
(gdb) li
(gdb) print pwd
```
gcc gdb2.c
gdb ./a.out
(gdb) run
[...]
(gdb) show environment BUFSIZ
[...]
(gdb) p num
[...]
make(1)
**make(1)**

*make(1)* is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
make(1)

make(1) is a command generator and build utility. Using a description file (usually Makefile) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
**make(1)**

`make(1)` is a command generator and build utility. Using a description file (usually `Makefile`) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs selective rebuilds following a dependency graph
**make** is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined
**make(1)**

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined
- different versions of `make(1)` (BSD make, GNU make, Sys V make, ...) may differ (among other things) in
  - variable assignment and expansion/substitution
  - including other files
  - flow control (for-loops, conditionals etc.)
make(1)

$ cd make-examples
$ ls *.?[ch]
cmp.c     ls.c     main.c     stat_flags.c     util.c
extern.h  ls.h     print.c     stat_flags.h

```
$ cd make-examples
$ ls *.?[ch]
cmp.c     ls.c     main.c     stat_flags.c     util.c
extern.h  ls.h     print.c     stat_flags.h
```

![Dependency Graph]

Lecture 10: Things That Will Make Your Life Significantly Easier  September 30, 2019
```
$ cd make-examples
$ ls *.ch
cmp.c  ls.c   main.c  stat_flags.c  util.c
extern.h ls.h   print.c  stat_flags.h
```
make(1)

$ cd make-examples
$ ls *.[ch]

cmp.c    ls.c    main.c    stat_flags.c    util.c
extern.h  ls.h    print.c    stat_flags.h
make(1)

$ cd make-examples
$ ls *.[ch]
cmp.c   ls.c   main.c   stat_flags.c   util.c
extern.h  ls.h   print.c   stat_flags.h
```sh
$ cd make-examples
$ ls *
chksum.c  ls.c  main.c  stat_flags.c  util.c
extern.h  ls.h  print.c  stat_flags.h
```

![Graph of dependencies between files]

**make(1)**
make(1)

$ ln -s Makefile.1 Makefile
$ make # or: make -f Makefile.1
[...]
$ make
[...]
$ make clean
$ export CFLAGS="-Wall -Werror"
$ make
[...]
$ make clean
[...]
$ make showvars
[...]
$ make CFLAGS="$\{CFLAGS\}" showvars
[...]

Repeat with other Makefiles.
Priority of Macro Assignments for `make(1)`

1. Internal (default) definitions of `make(1)`
2. Current shell environment variables. This includes macros that you enter on the `make` command line itself.
3. Macro definitions in `Makefile`.
4. Macros entered on the `make(1)` command line, if they follow the `make` command itself.
Ed is the standard text editor.

$ ed
?
help
?
quit
?
exit
?
bye
?
et flaming death
?
^C
?
^D
?
Ed is the standard text editor.

$ ed
a
ed is the standard Unix text editor.
This is line number two.
.
2i
.
%l
3s/two/three/
w foo
q
$ cat foo
**diff(1) and patch(1)**

**diff(1):**

- compares files line by line
- output may be used to automatically edit a file
- can produce human “readable” output as well as diff entire directory structures
- output called a *patch*
**diff(1) and patch(1)**

**patch(1):**
- applies a `diff(1)` file (aka *patch*) to an original
- may back up original file
- may guess correct format
- ignores leading or trailing “garbage”
- allows for reversing the patch
- may even correct context line numbers
**diff(1) and patch(1)**

```bash
$ diff Makefile.2 Makefile.5
[...]
$ cp Makefile.2 /tmp
$ ( diff -e Makefile.2 Makefile.5; echo w; ) | ed Makefile.2
$ diff Makefile.[25]
$ mv /tmp/Makefile.2 .
$ diff -c Makefile.[25]
$ diff -u Makefile.[25] > /tmp/patch
$ patch </tmp/patch
$ diff Makefile.[25]
```
diff(1) and patch(1)

Difference in ls(1) between NetBSD and OpenBSD:

$ diff -bur netbsd/src/bin/ls openbsd/src/bin/ls

Difference in ls(1) between NetBSD and FreeBSD:

$ diff -bur netbsd/src/bin/ls freebsd-ls/ls
Revision Control

To be continued...
Links

GDB:
https://sourceware.org/gdb/current/onlinedocs/gdb/

http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html